Robot Navigation using Deep Reinforcement Learning

***Abstract*— The aim of this project is to navigate a robot autonomously in a dynamic environment by avoiding obstacles and reach the end goal using deep reinforcement learning. Learning to navigate in an unknown environment is a crucial capability of mobile robot. Conventional method for robot navigation consists of three steps, involving localization, map building and path planning. However, most of the conventional navigation methods rely on obstacle map, and don't have the ability of autonomous learning. In contrast to the traditional approach, the experiment results show that the robot can reach to the desired targets without colliding with any obstacles. We demonstrate enhanced results in terms of safety and robustness over a traditional baseline approach based on the dynamic window approach.**

Keywords*— *Deep reinforcement learning, autonomous navigation, TD3 DDPG,

I. Introduction

In this paper, we study about the navigation problem in the dynamic environment and collision avoidance with obstacles with deep reinforcement learning. The goal is to find a control policy for an agent to navigate through a simulated indoor environment and reach the destination without colliding with obstacles. Navigation in an unknown environment is a reinforcement learning (RL) problem because the best navigation plan can only be discovered through trial-and-error interaction with the environment.

With the recent developments in Deep reinforcement Learning (DRL) for autonomous robot navigation high precision decision making is now possible by autonomous agents. Using DRL, a policy can be learned to control an agent in order to complete the objective job in an unknowable environment. Without making a prior plan, a robot can navigate the environment by applying learnt policy to an input. The generalization problem in DRL is addressed in several works, with various solutions proposed. These solutions include increasing the similarity between training and execution environments with data augmentation, domain randomization, or creating different environments throughout training, and handling differences between environments with conventional regularization techniques. However, in order to cover all properties of various contexts, these generalization algorithms necessitate extensive data variety and training steps. Even said, algorithms are still capable of discovering issues when applied in settings other than those used for training.

A crucial component of the conventional navigation system is path planning. This module can be used in a variety of ways depending on the amount of environmental data that is categorized as either global or local path planning. The process of choosing a complete path using a known environmental map is known as global path planning. The A-star, ant colony optimization, and rapid exploration random tree are often employed techniques; however, because they depend on well-known static maps, they are challenging to apply in dynamic environments. The Artificial Potential Field (APF) and the dynamic window approach are two local route planning techniques that are used to deal with dynamic changes in the environment and replan local paths.

In this project, we proposed an autonomous navigation system that combines path planning with honed reactive navigation in order to reach a specified goal. A preliminary navigational plan is computed to reach the chosen intermediate target. Then, local reactive navigation is carried out using the learned policy from DRL.

The remaining of this paper is organized as follows: In Section II methods are discussed, in section III Implementation and simulation is discussed and in section IV results are given, Finally, this paper is concluded with section V.

II. Methods

Environment

The task that we are attempting to do must be understood before we begin the actual motion policy training of a mobile robot. The main aim of this project is to "identify the most effective sequence of actions that guide the robot toward a specified goal." However, how can we turn this issue into something that a machine can comprehend and handle? Both the action and the environment that the action responds to must be taken into account. It is simple to translate an action into a mathematical representation when used in a mobile robot environment.

Environmental states must be observed logically and numerically in order to elicit a response. The use of a LiDAR or laser sensor is one of the most basic methods for robotics applications to sense its surroundings. The reasoning for this is straightforward: the closer an object is to the laser sensor, the quicker the light emitted by the laser will return to the photoreceptor. The distance to the object can then be calculated directly using this speed. We can assume a robot will make judgments based on the distance it is from obstacles, and we can infer a mobility policy from that. This provides a straightforward numerical representation of the surroundings of the laser sensor and subsequently the robot.

The robot's ability to determine its location is a final point to take into account. The robot must be told this knowledge since we have a certain goal in mind for it to accomplish. This information should be provided centered on the robot's situation in order to learn an ego policy. The distance to the goal position and the angle difference between the robot heading and the heading in the direction of the objective will be used to describe this in polar coordinates.

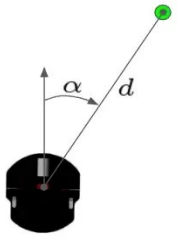

Figure 1: Representing how robot reaches its goal.

We may now characterize our condition and action because the environment and the agent have been expressed numerically. Action is a tuple, as was mentioned:

$$a = (v, \omega)$$

However, things become a little bit more challenging for the state. The environment's surroundings must first be modeled. We'll use the laser readings to accomplish this. The objective will then be specified in polar coordinates. To help the neural network understand how quickly we are already moving, we might also want to provide it with information on the robot's internal status right now. As a result, we will include the action taken at the preceding time-step in our representation of the state. Considering this, our final state representation will be a 1D vector made up of:

$$s = \left(laser_{state} + polar_{goal_{coordinates}} + previous_{action}\right)$$

The policy is rewarded based on the following reward function:

$$r(s_t, a_t) = \begin{cases} r_g & if\ D_t < \eta_D \\ r_c & if\ collison \\ v - |\omega| & otherwise, \end{cases}$$

Here, r is the state-action pair (s_t, a_t)at timestep t, which is dependent on three factors. A positive goal reward r_g is applied if the distance to the target at the current timestep D_t is lower than the threshold η_D. A negative collision reward r_c is applied if a collision is discovered. If neither of these circumstances holds true, a reward is given right away depending on the present linear and angular velocities. A delayed attributed reward approach is used to direct the navigation policy towards the specified target and is based on the following calculation:

$$r_{t-i} = r(s_{t-i}, a_{t-i}) + \frac{r_g}{i}, \qquad \forall_i = \{1,2,3, \dots, n\}$$

where n denotes how many steps have already been taken where rewards would have been adjusted. As a result, the positive goal reward will be distributed decreasingly over the previous n steps in addition to the state-action pair at which the objective was achieved. The network developed a local navigation strategy that can reach a local objective while also avoiding obstacles based solely on laser inputs.

Immediate reward in its basic form can be expressed as:

$$r = v - |w|$$

The reasoning behind it is that the robot must understand that it must move around rather than remain still. Robot learns that traveling forward is beneficial and spinning is bad by initially setting a positive incentive for linear motion. Even though it frequently crashes in the beginning, the episode payout is still greater than simply turning while seated. It quickly discovers that, although taking a turn close to an obstacle may still be harmful, it is less harmful than crashing. Because the turning penalty is reduced the smaller the rotation, the robot quickly learns to avoid obstacles with a smooth motion. The robot will eventually discover that the advantages of achieving the goal exceed the disadvantages of turning while randomly moving through the surroundings. The robot will accidentally search for a route out of the pocket even if it ends up in one because it knows that moving forward will result in a reward. The robot will, however, prefer to also have linear motion while having rotational motion as that would give a better reward, thus this strategy does not give you the fastest path to the goal. In order to turn around, the robot will therefore, if at all possible, make a larger turn, lengthening the overall trip.

If the robot comes within one meter of any obstacle, there will be an additional negative reward. The robot becomes weary of impediments in general when this "repulsion" is used, and it moves around them with a wider gap.

III. IMPLEMENTATION AND SIMULATION

To achieve autonomous navigation in an unknown environment we propose an autonomous navigation structure with deep reinforcement learning-based local navigation. At every step, the destination goal point is given to the robot in the form of the x and y coordinates relative to its current location. Local motion is carried out in a planning-based navigation stack after the local planner. With our method, learnt policy takes the place of this navigation stack layer. we use DRL to train the local navigation policy in a simulated environment.

The neural network architecture for the navigation policy is DDPG-based. An actor-critic network called DDPG enables for actions to be taken in continuous action space. Laser readings in the 180-degree range in front of the robot provide a description of the immediate environment. This data is paired with the goal's polar coordinates relative to the robot's position. The actor-network of the DDPG uses the combined data as an input state s. There are two fully connected (FC) levels in the actor-network. Each of these layers is followed by activation of the rectified linear unit (ReLU). The last layer is then linked to the output layer via two action parameters, or a_1, and a_2, which stand for the robot's linear velocity and angular velocity, respectively. The output layer is limited in the range (-1, 1) using a hyperbolic tangent activation function. The maximum linear velocity v_{max} and the maximum angular velocity ω_{max} are scaled before the action is applied to the environment as follows:

$$a = \left[v_{max}\left(\frac{a_1}{2}\right),\ \omega_{max}\ a_2\right]$$

Backwards motion is not considered because the laser readings only capture data in front of the robot, hence the linear velocity is changed to only be positive. In the critic-network, the Q value of the state-action pair $Q(s, a)$ is assessed. A pair of the state s and actions a is provided as input to the critic-network. A completely linked layer receives the state s before activating ReLU with output l_s.

Two distinct transformation fully connected layers (TFC) of the same size τ_1 and τ_2 receive both the output and the action from this layer. Then, these layers are brought together as follows:

$$l_c = l_s W_{\tau_1} + a W_{\tau_1} + b_{\tau_2}$$

Where l_c is the combined fully connected layer (CFC), $W_{\tau_1}$and $W_{\tau_2}$are weights of the τ_1 and τ_2 respectively. b_{τ_2} is the bias of the layer τ_2. The merged layer is next activated with ReLU. Following that, it is linked to the output via a parameter that represents the Q value. Figure 2 depicts the entire network architecture in detail.

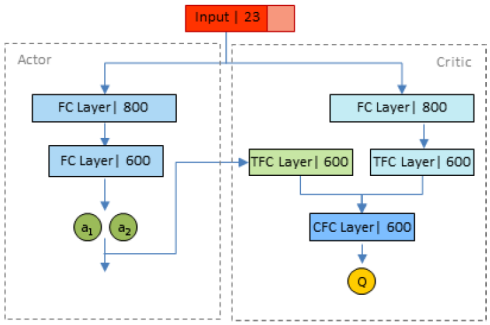

Figure 2: Actor and Critic parts of DDPG network.

Similar to DDPG, TD3 is an actor-critic network. This indicates that there are two networks: a "actor" network that determines what action to take, and a "critic" network that determines how effective that action is. To put it simply, the TD3 architecture addresses the issue of overestimating the Q-value by extending the DDPG architecture. To achieve this, a second critic network is added to the loop, and the output from the one that yields lower Q-value estimates is chosen.

Therefore, an actor-network should be created that uses the environment's state as an input and output to determine what the robot should do. The environmental state and the action from the actor network must also be created in two critic networks, which will produce the estimated value of this state-action pair.

Actor Network

We need to develop a decoder that will translate state input to action in order to construct an actor model. We develop an actor class with three completely connected, linear layers. We specify the layers we'll utilize in the forward pass during class initialization. A single-dimension vector reflecting the ambient state will be the input to the first layer. This state will have 800 parameters embedded in it. Since the layers will be called one after the other, we need to ensure that the number of parameters in each layer is the same. As a result, the 800 parameters needed for the following layer input must remain the same.

Then, they will be mapped to 600 parameters. The 600 characteristics will be immediately mapped to the amount of actions that our robot is capable of performing in the final layer. In our situation, we will be in charge of a ground-moving robot with adjustable linear and angular velocities. Our robot can only move at certain maximum and minimum speeds, hence the neural network's output must be constrained to a min-max range. By defining a Tanh activation function, we can do this. Tanh function will limit the output to values between -1 and 1. We specify the sequence, how the layers will be called, and how the values will be sent through it in the forward pass. We first call the first defined layer, activate it with ReLU, and then repeat the process for the second layer. The output is then forced to be in the range of -1 to 1 by calling the third linear layer and using the Tanh function. The output of this straightforward actor network will serve as the foundation for our robot's linear and angular velocities.

Critic Network

Similarly, we'll establish a network of critics. As previously stated, we need two critics to reduce the overestimation of the Q-value. Fortunately, since both of these critics will always be active at the same time, only one critic class needs to be used to define both.

The decoder for the Q-value in this case is fairly similar to which it is used in actor network. We will require both the state information and the action from the actor-network as inputs in our critic network since we need to evaluate not the state but rather the actor's response to it. The state-action pair can be defined in a variety of ways as an input. For instance, putting the action values at the end of the one-dimensional vector representing the state. Then we could use the same form of decoder we saw in the actor network. For the critic network, we essentially treat the state and action as two independent inputs. Using a single linear to embed the state data, pass through the state embedding and action values individually in the second layer, combine them, and get our value estimation in the third layer. We must define these layers twice in the initialization because we will establish both critic networks with a single class.

We specify the propagation's path in the forward pass. First, the first layer is solely passed through by the state input, which is then given ReLU activation. The second layer is then passed over it. The output size of the single layer through which the action values are likewise simultaneously transferred has the same size. The weights of the corresponding second layers are multiplied by state embedding and action. A tensor with 600 parameters of the same size will be produced by each multiplication. The bias of the second action layer can then be added to these tensors as well as their total. In practice, this technique for incorporating action inputs into the neural network has been successful. The combined state-action pair embedding is then mapped to the Q-value estimation using a final layer. Because the Q-value has no minimum or maximum, there is no need for a capping function in this case. We will receive the Q-values for both critic networks as output.

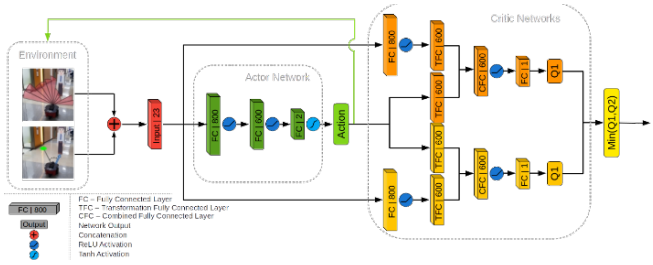

Figure 3: TD3 Network

TD3 Network

TD3 network is implemented with the combination of both Actor and Critic networks. In TD3 Network, *Soft update* is implemented that will be used to select a subset of data called 'batch' which will optimize network parameters. However, this does not imply that we will have identified the ideal parameters for every scenario. In fact, using these optimized parameters directly will cause our learning to become very unstable because the network parameters will constantly be changing as a result of optimization for new sets. For this reason, each of our networks will have a control or target

network. We will improve our base network and get new parameters on each training cycle. The base network will be reset to have the same parameters as the recently updated target network after we inject a tiny portion of these optimized base network parameters into the target network.

In this way, the training process will remain stable while the target network is gently pushed (hopefully) in the general direction of the optimal policy with each update. This is the rationale behind the necessity of creating actor target and critic target methods in addition to calling our actor and critic classes. After that, we load any previously saved parameters for each class and configure the Adam optimizer for each one. Next, we build a writer to record our data for Tensorboard visualization, set the maximum action value, and set a training iteration counter.

We also build a replay buffer, which will preserve the simulation-related experiences the robot has faced and serve as a dynamic dataset for the neural network's optimization. We select a batch from the replay buffer's collection of recorded robot motion experiences. Following that, we add some noise to the next potential action for each state in the batch. Then, using the target networks to predict the potential Q values for this subsequent state-action combination for both critic networks, choose the sample from the batch with the smallest value of both outputs. We use the Bellman equation to calculate the goal Q value from the minimal values. The Q values of base critic networks are then obtained. We may determine how far off our base network is by computing the mean squared error between the base network critic values and the goal Q value. The tricky part of this is that we are discounting all future state rewards while estimating the current state reward using the base network. Target Q value, however, uses an estimate for future rewards and already-known current state rewards. Therefore, calculating the loss against the target value should nudge the network in the direction of proper optimization. The target value should be closer to the real estimation. The critic network is then simply optimized using gradients in the loss. Additionally, we collect the average and maximum Q values in the av Q and max Q variables.

The Delayed component of Twin Delayed DDPG (TD3) is done by performing this step just once every policy repetitions, we essentially postpone the updates of the actor network when compared to the critic network as well as the soft updating of network parameters. This should provide better actor parameter updates and boost network stability as critic would be more precise. We determine the Q value for the state-action pair during the actor update. We adjust the network parameters in a way that will produce the maximum Q value feasible. It may seem unusual that we first obtain the critic network loss using fixed actor parameters and then utilize the same critic to determine the Q value for actor updates. The repeated "pushing" of each network's policies back and forth may make this process unstable, but it finally moves in the right direction, giving us the (hopefully) best actor and critic networks. Next, we simply apply a gentle update to both network parameters. To display our Q and loss value variables in Tensorboard, simply record them. A TD3 neural network that can be used to learn robot movements should be produced as a result.

Training

Now that everything has been set up, we can begin our training loop. Although other techniques (such as the maximum number of epochs, elapsed time, etc.) can also be simply set up, we will base the amount of training on the maximum training steps.

Episode - A series of consecutive actions until one of the termination conditions is met.

Epoch - A period of time between performing evaluations determined by the evaluation of frequency option.

The network training won't occur on the first execution of the function since the timestep value will still be zero, as we do not have anything yet to train on. However, a neural network training will be conducted when each succeeding episode is finished. Therefore, we train the neural network following each episode. The value we supply in for episode timesteps controls the training cycles. The argument goes that if the episode had more timesteps, we would have gathered more fresh samples and had more fresh data to optimize for. There will be fewer training cycles if there aren't many fresh samples.

After that, we determine whether it is appropriate to evaluate the existing model. If so, we carry out the assessment, note the outcomes, and save our model. After these inspections are finished, we must reset our surroundings. This will start a new episode, give the robot a new random beginning location and orientation, and give a new random target position. We will be able to observe the new environment from the reset state. Following that, we simply reset our counting variables to 0 and begin running a new episode.

It is typical in DRL to use a greedy approach. In other words, the neural network's job is to choose the best-calculated result for any input. Unfortunately, this could lead to a vicious loop during training as one first determines a positive conclusion, then follows it and reinforces the positive outcome without considering any potential better options. This is frequently described as an exploitation technique. We need to compel the neural network to consider all potential outcomes in order to find the ideal result that is exploration strategy. Adding random noise to the neural network output and then observing the outcome in the surroundings is a typical technique for doing this.

We determine how much noise we want to add to the neural network's action calculation. Over the exploratory decay steps, the noise level will drop until it meets the exploratory min value. We use a neural network to calculate the action, and then we add noise to this value to update it. Robots are physically limited in both their highest and minimum velocities, though. To get the final action value to test in the environment, we cut the action value in this range, random value is added to the action. As a result, we receive a random number with a Gaussian distribution. As minus values are equally likely to be supplied as positive ones, it follows that if we were to average out the noise's mean value, it would be extremely close to 0. For the robot to escape a precarious situation, however, it must take swift action and a random Gaussian noise applied to each step is insufficient. Let's say the robot has come face to face with an obstruction as demonstrated in the below figure.

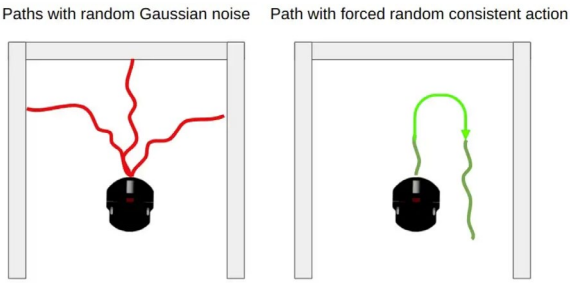

Figure 4: Representing robot action under Gaussian noise and forced random consistent action.

If we decide to employ this strategy, we might not want to do so for each interaction that occurs close to an impediment. As a result, we only use this approach if the random value we query is greater than 0.85. Additionally, we determine if the robot is closer to the obstruction in front of it than 0.6 meters. Keep in mind that our state is represented by the values 20 + 2 + 2, the first 20 of which are the laser sensor state. We check the values from 4 to 16 in our state representation to see if an obstacle is in front of the robot and not on the sides. Additionally, we look to see if this method's sequential execution has already locked us in. If everything checks out, we choose a random number between 8 and 15 for the number of steps this random activity will take. then take a random action sample. Update the action value, then decrement the random action counter. In this case, we set $action[0] = -1$ to have the robot rotate only randomly and in order to set a completely random action.

The next step is to "fix" the action value. In the range of $-1, 1.$, our TD3 neural network model generates two action values. However, the field of view for our laser sensor is limited 180 degrees in front of the robot. The robot has no way of knowing what is behind it, therefore it would be reckless of us to allow it to move backward. Our robot must therefore have a minimum linear velocity of 0 meters per second. When the action represents the linear velocity as we squeeze it in the range of $(0, 1)$, we gain two Boolean values of whether the state completed and if the target was reached along with the state-action pair reward and a new state. To our gazebo simulator environment, where it is executed, we can pass this value.

Then, we determine if the maximum number of steps has been reached and update the values for done and done bool accordingly, in Python int and bool values can be used interchangeably, but here we use int as we use the numeric value in the Bellman equation as the terminality of the state. Then, just update the training counters and the current running state for which we will calculate the action value in the following iteration of the training loop. Until it is finished, this loop will continue to run, calculate actions for each state, collect data, and train the neural network model. Alternatively, we may force it to stop by pressing CTRL+C on the terminal. We save the evaluation data and the final network settings after the loop is complete.

How to do an evaluation is a final topic that we have not yet covered. We provide a unique evaluation function for this. Remember that every period ends with the calling of this. Set some settings, such as average reward and collision rate, that will gather the data we might be interested in. Then we begin the evaluation loop, which will carry out the number of random evaluation runs. The procedures are the same as during training, with the exception that we are interested in how well the present model works and do not apply random noise to the estimated actions. We also note the reward and whether or not robot crashed during the run. We may use the reward to determine whether a collision happened because the hardcoded reward for a crash is -100. Alternatively, we can look if done is True while target is False. We determine the average reward and collision rate after completing all of the evaluation runs. then use the terminal to print them out. In a perfect world, we would observe a high positive average reward and near to zero collision rate. We provide the average incentive, which will be included in the evaluation findings.

The neural network model that trains a mobile robot motion policy from laser, goal, and robot motion inputs has now completed its training. Any simulation environment might theoretically be utilized. It is not required to be our Gazebo simulator or a laser sensor input that captures environmental data. Simply assigning a different simulation environment or changing the environment would enable the use of a sensor with a different number of inputs.

IV. RESULTS

The learning of local navigation using deep reinforcement learning is formed on a computer with Intel Core i7-11700H CPU, 16 GB RAM, and a single NVIDIA RTX 3050 Mobile GPU. The training of the neural network was carried out in the Gazebo simulator and controlled by the Robot Operating System (ROS) commands. We trained the network for 800 episodes which took approximately 8 hours. Each training episode consists of 500 steps or until a collision. v_{max} and ω_{max} were set as 0.5 meters per second and 1 radian per second, respectively. The delayed rewards were updated over the last n = 10 steps. The training was carried out in a simulated 10x10 meter-sized environment as shown in the Figure.

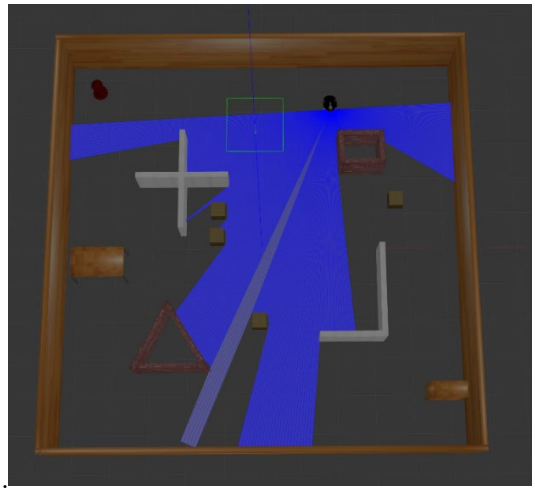

Figure 5: Gazebo simulation.

In the above figure, the blue area represents the laser senser reading, this is how robot learns about the environment and the green square box represents the goal point and the obstacles are all scattered around the environment making the environment dynamic as these box shaped obstacles changes its location within environment every time after the robot resets its position.

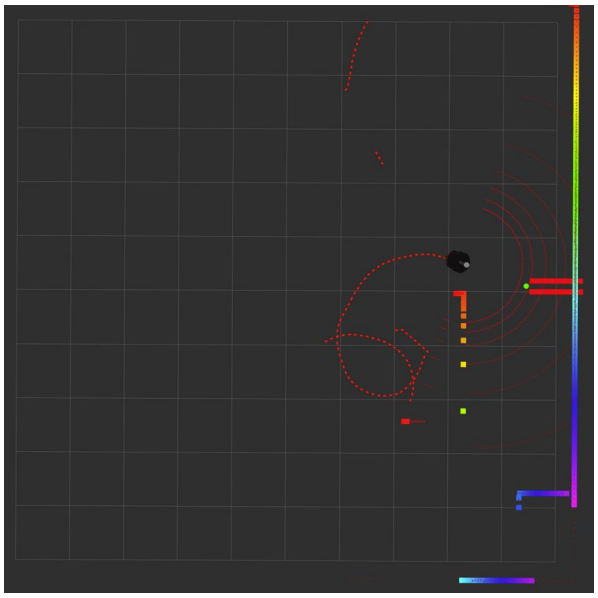

Figure 6: simulation in Rviz

In this figure, we can observe that, the robot is moving towards the goal to the green dot, while RGB colors represent the level of danger of hitting the obstacle resulting avoiding the collision. The robot learns to avoid collisions after nearly 100 episodes, and successfully avoids collision at an accuracy of nearly 98% after 300 episodes of training.

V. DISCUSSION & CONCLUSIONS

. As the experiments show, the system successfully performs navigation. Moreover, the task of introducing a neural network-based module for an end-to-end system proves to be beneficial as it allows the robot to move without generating an explicit plan, but its shortcomings are alleviated by the rest of the navigation system. The obtained experimental results show that the proposed system works reasonably close to the optimal solution obtained by the path planner from an already known environment. After successful training and testing of a neural network. The robot can finally avoid the obstacle collision avoidance in a dynamic environment.

VI. REFERENCES

[1] R. Parr and S. J. Russell, “Reinforcement learning with hierarchies of machines,” in Advances in neural information processing systems, 1998, pp. 1043–1049.

[2] T. G. Dietterich, “Hierarchical reinforcement learning with the MAXQ value function decomposition,” J. Artif. Intell. Res.(JAIR), vol. 13, pp. 227–303, 2000.

[3] H. Van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-Learning,” in AAAI, 2016, pp. 2094–2100

[4] Learning to Navigate through Complex Dynamic Environment with Modular Deep Reinforcement Learning by Yuanda Wang, Haibo He, and Changyin Sun.

[5] Goal-Driven Autonomous Exploration Through Deep Reinforcement Learning by Reinis Cimurs, Il Hong Suh, and Jin Han Lee. https://arxiv.org/pdf/2103.07119.pdf

[6] Deep-Reinforcement-Learning-Based Semantic Navigation of Mobile Robots in Dynamic Environments by Linh Kästner, Cornelius Marx and Jens Lambrecht.

[7] Robot Navigation with Map-Based Deep Reinforcement Learning Guangda Chen, Lifan Pan, Yu’an Chen, Pei Xu, Zhiqiang Wang, Peichen Wu, Jianmin Ji∗ and Xiaoping Chen.

[8] Deep Reinforcement Learning Based Mobile Robot Navigation: A Review by Kai Zhu and Tao Zhang.

[9] https://github.com/ibrahimgb/robot-navigation-using-deep- reinforcement-learning.